
# Simulation d'une équipe de robots pompiers

Equipe 37

Novembre 2022

# 1 Inroduction

Le TP en question consiste à développer en Java une application pour simuler une équipe de robots pompiers opérants de façon autonome en milieu naturel. Le sujet comprend quatre parties de difficultés diverses. En fait, le but des première et deuxième parties était de mettre en place les classes de base afin de modéliser le problème. Les deux tests ***TestScenarioZero*** et ***TestScenarioUn*** ont permis de tester un ensemble de scénarios et de tester l'opération des robots sur les cartes. Puis, dans la troisième partie, nous affinons les mouvements des robots en utilisant des trajets plus courts pour être plus rapides et plus efficaces. Enfin, les robots reçoivent l'ordre d'un chef des pompiers pour éteindre tous les incendies. Dans ce qui suit, on vous donnera notre algorithme avec plus de détails.

# 2 Représentation des données

## 2.1 Classes de bases

Pour représenter les données du problème, nous avons implémenté l'ensemble des classes citées dans le sujet à savoir la classe ***Carte***,la classe ***Case*** et la classe ***Robot***. La classe Robot est de type abstrait, ainsi elle encapsule l'ensemble des attributs et méthodes qui peuvent être utilisés par ses sous-classes, classe ***Drone***, classe ***Roues***, classe ***Chenilles***, et classe ***Pattes***.

Par la suite nous avons relié toutes les classes qui sont en relation en ajoutant à l'une les attributs de l'autre. Par exemple, la classe robot contient un attribut carte qui représente la carte dans laquelle se trouve le robot ainsi qu'un attribut case pour indiquer sa position. Ensuite, nous avons modifié la classe ***lecteursDonnees*** pour ajouter en plus de la lecture du fichier (map), l'instanciation d'une nouvelle classe créée ***DonneesSimulation*** qui regroupe l'ensemble des données de problème: la ***carte***, les ***robots*** et les ***incendies***.

## 2.2 Interface graphique et simulation

A l'aide de l'interface graphique fournie ***Gui.jar***, on a implémenté la classe ***Simulateur*** qui réalise l'interface ***Simulable*** afin d'afficher les données et mettre à jour l'affichage en fonction de l'état des données.

# 3 Gestion des évènements:

Concernant la simulation de scénario, nous avons décidé de créer une classe abstraite *evenement* munie de l'attribut *date* représantant la date ou l'évènement a lieu, ie la date où les données de la simulation doivent être modifiées, ainsi qu'une méthode *execute()* qui modifie les données de la simulation. Nous avons décidé de stocker l'ensemble des évènements à exécuter à l'aide d'une PriorityQueue, ainsi les évènements de la simulation sont stockés et triés selon leur date d'exécution, ainsi nous aurons rapidement accès au prochain élément à exécuter. A chaque fois qu'un évènement est exécutée nous le supprimons de l'ensemble des évènements. Il faudra faire attention à ajouter des évènements dont la date d'exécution n'a pas encore été atteinte, ainsi la méthode *ajouteEvenement(e: Evenement)* de simulateur lancera une exception si cette contrainte n'est pas respectée. *Vous trouverez dans le répertoire test deux fichier (Scenario0.java et Scenario1.java) représentant les deux scénario de test décrit dans le sujet (2.2 Scénarios de test p.5).*

# 4 Optimisation du simulation: Plus Court Chemin

Pour programmer un déplacement d'un robot nous devons d'abord calculer le plus court chemin (le temps de trajet) pour se rendre sur une case donnée en fonction de ses propriétes et ses caractéristiques de terrain . Puisque les coûts sont positifs nous allons utiliser l'algorithme de Dijkstra pour trouver une arborescence (les itinéraires des plus courts dans un graphe). Nous avons modélisé le graphe avec la carte et les sommets d'un graphe avec les cases pour ne pas définir plusieurs de classes et pour centraliser le calcul. La case stocke le plus court chemin (LinkedList a partir d'une case source) calculé avec l'algorithme de Dijkstra et une attribut **distance** qui contient le coût

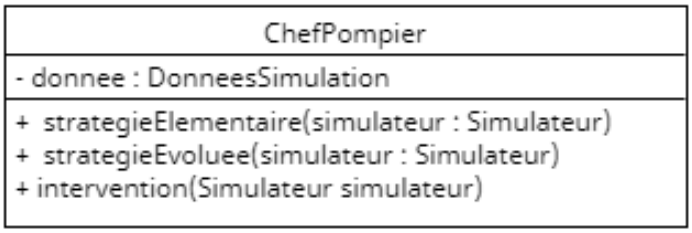


Figure 1: Diagramme présentant la classe *ChefPompier* et les méthodes associées

(initialisé au début avec infini). L'algorithme de Dijkstra est implémenté dans la classe *Carte* par la fonction *calculePlusCourtChemin(Robot robot)*. Il commence par initialiser la case de robot (case source) a une distance nulle, a chaque itération nous choisissons parmi les cases non-visitées celle de distance minimal (*getDistnaceMin(casesNonVisites*), nous la considérons comme case visitée. Ensuite nous mettons à jour les distances des cases voisines de la case sélectionnée a l'aide de la fonction *calculDistanceMinimale*. La mise à jour s'opère comme suit: la nouvelle distance du sommet voisin est le minimum entre la distance de la case et celle obtenue en ajoutant le coût de déplacement a cette case (en utilisant la méthode *calculTempsDeplacement*) et la distance de la case sélectionnée. Nous continuons ainsi jusqu'à l'épuisement des cases non visitées. A chaque fois que nous voulons exécuter l'algorithme de Dijkstra, nous devons exécuter la méthode *reset( )* de la classe *Carte* pour initialiser les distances des case a plus infini et vider les plus court chemins.

# 5 Résolution du problème:

Après l'implémentation des classes et méthodes modélisant l'environnement de travail (Robot, incendies, case, calculePlusCourtChemin,...), C'est le temps pour définir une stratégie d'affectation des tâches aux robots. On a définit la classe *ChefPompier* qui prend *DonneesSimulation* comme seul attribut puisque le chef pompier a une vision d'ensemble de la situation. Cette classe implémente deux méthodes en fonction de la stratégie mis en oeuvre:

## 5.1 Stratégie élémentaire:

Cette stratégie consiste à affecter chaque incendie non affecté a un robot non occupé qui se rend vers l'incendie avec un plus court chemin. Le robot sélectionné programme la séquence d'évènements nécessaires pour réaliser son déplacement, et son état devient occupé. Nous calculons le temps de déplacement pour bien programmer l'évènement de *VerseaEau* a la date adéquate. Un robot est occupé si il est en train de se déplacer, éteindre un incendie ou bien son réservoir est vide. L'incendie devient non affecté si le robot sélectionné n'a pas une capacité de réservoir nécessaire pour l'éteindre. Le chef pompier refait le même processus sur l'ensemble des incendies non affectés chaque 100 pas de temps, la sous-classe *InterventionChef* de la classe *Evenement* permet d'exécuter périodiquement la stratégie de chef pompier a condition qu'il y est un incendie non éteint et accessible par au moins l'un des robots.

## 5.2 Stratégie un peu plus évoluée:

Dans cette stratégie, nous cherchons a optimiser le processus global d'affectation des robot. Le chef pompier propose à chaque incendie non affecté a tous les robots non occupés avec un réservoir non vide. puis il sélectionne le robot le plus proche a cet incendie en calculant le coût (temps) de chaque chemin. Le robot sélectionné programme la séquence d'évènements nécessaires pour réaliser son déplacement et verser l'eau. Le chef pompier attend un certain laps de temps (100 next) puis il exécute de nouveau la stratégie a condition qu'il y est un incendie non éteint et accessible par au moins l'un des robots.